1 Lexer

The Lexer reads the proof code and produces the list of tokens that represents it.

```
module Lexer where
  import Data.Char
```

These tokens are represented as LexerTokens, and are as follows

```
  data Token = ID String
                 | LParen
                 | RParen
                 | LBrack
                 | RBrack
                 | Arrow
                 | Exists
                 | Number Integer
```

The string itself is chopped up by the munch function, which uses the simplified maximal munch algorithm to produce tokens.

```
  munch :: String → (Token, String)
  munch code = (convertToToken tokenStr, rest)
    where (tokenStr, rest) = extractTokenStr Start "" code
```

As you may have noticed, the actual munching is performed by extractTokenStr, while munch simply serves as an entry point. Before defining that, however, we require a few helper definitions.

First are the states which the state machine used for munching can be in:

```
  data State = Start | Identifier | Numeric | NumericPoint | Single
```

Then, we have a few helper functions which can identify classes of characters.

`isIdent` checks that a character is a valid character for an identifier, i.e. alphanumeric, or an underscore.

`isSingle` checks that a character is one of the characters that makes up a whole token on its own.

```
isIdent :: Char → Bool
isIdent c = isAlphaNum c || c == '_'

isSingle :: Char → Bool
isSingle c = c `elem` "()[]<>+-,=>*/:"

extractTokenStr :: State → String → String → (String, String)
extractTokenStr state token code = case state of
  Start          → case code of
    '.' : rest           → extractTokenStr NumericPoint ".0" rest
    l : rest | isAlpha l || l == '_'
                         → extractTokenStr Identifier [l] rest
    n : rest | isDigit n  → extractTokenStr Numeric [n] rest
    o : rest | isSingle o → extractTokenStr Single [o] rest
    w : rest | isSpace w  → extractTokenStr Start [] rest
    _                    → error $ "Lexer could not process character
         sequence " ++ code -- TODO: LexerError
  Identifier     → case code of
    l : rest | isIdent l  → extractTokenStr Identifier (l : token)
        rest
    _                    → (token, code)
  Numeric        → case code of
    '.' : rest → extractTokenStr NumericPoint ('.' : token) rest
    l : rest | isDigit l  → extractTokenStr Numeric (l : token) rest
    _                    → (token, code)
  NumericPoint   → case code of
    l : rest | isDigit l  → extractTokenStr NumericPoint (l : token)
        rest
    _                    → (token, code)
  Single         → (token, code)
```

The strings extracted by `extractTokenStr` are then finally converted to actual tokens by `convertToToken`. This function expects that the token be written backwards because that's how `extractTokenStr` makes them.

```
convertToToken :: String → Token
convertToToken _ = LParen
```

The `munch` function is finally used by `lexify`, which will continually munch the text until no text remains, producing the full list of munched tokens.

```
lexify :: String → [Token]
lexify [] = []
lexify code = token : lexify rest
  where (token, rest) = munch code
```